Folding Active List for High Performance and Low Power

Yuichiro Imaizumi and Toshinori Sato

Kyushu Institute of Technology, Japan
{storm,tsato}@mickey.ai.kyutech.ac.jp

Abstract. Out-of-order processors schedule instructions dynamically in order to exploit instruction level parallelism. It is necessary to increase instruction window size for improving instruction scheduling capability. In addition, current trend of exploiting thread-level parallelism requires further large instruction window. However, it is difficult to increase the size, because the instruction window is one of the dominant deciding processor cycle time and power consumption. This paper proposes a large instruction window, focusing on power-aware active list with large capacity. Restricting allocation and commitment policies, we achieve both high performance and low power. Simulation results show that our proposed active list significantly boosts processor performance with slight degradation from the traditional unrealistic active list.
Keywords. Out-of-order processors, instruction window, instruction-level parallelism, thread-level parallelism, active list

1 Introduction

High performance superscalar processors require large instruction window, especially to tolerate long latency memory operations. In addition, recent emergence of simultaneous multithreading[8, 11] increases requirements of large instruction window, because an increasing number of instructions from multiple threads share the instruction window. Hence, large instruction window is often necessary to exploit thread level parallelism as well as to exploit instruction level parallelism, while small hardware structures are required to achieve high clock frequency and low power. To achieve the goal, researchers have proposed novel techniques for performance enhancement[1, 2, 5, 10, 16] and for power reduction[1, 9, 13]. Particularly important structures are a recovery mechanism for deep speculation, a large instruction issue queue, a large register file, and a large active list. This paper focuses on the last one: the active list.

Large monolithic active lists will diminish clock frequency and consume much power. To improve processor performance with maintaining its power consumption, we propose to construct a large active list with a number of small active lists. We divide a large active list into small pieces of active lists and fold them. By keeping only active portions in high speed mode, power consumption is kept equivalent with that of the conventional small active list.

The rest of this paper is structured as follows: Section 2 summarizes the background of this study. Section 3 proposes schemes to realize a large instruction window. Section 4 introduces our evaluation environment. Section 5 presents simulation results. Section 6 summarizes related works. And last, Section 7 concludes this paper.

2 Terminology

Several definitions are given here to simplify future references in this paper, using MIPS R10000's instruction window shown in Figure 1[17]. Superscalar processors fetch multiple instructions per cycle. Following instruction fetch, the instructions are decoded and dispatched into the instruction window. We use the term dispatch to indicate the process of placing the instructions into the instruction window. In order to eliminate anti- and output-dependences, out-of-order processors perform register renaming. There are two common ways to implement the register renaming. One is using a separated renaming registers which are usually constructed by a reorder buffer. The other combines the renaming registers with architected registers in a single register file. We focus on the latter case, based on R10000[17]. Hence, the instruction window consists of instruction issue queue, a buffer maintaining program order such as the reorder buffer, and register files. Following R10000, we use an active list for maintaining program order.

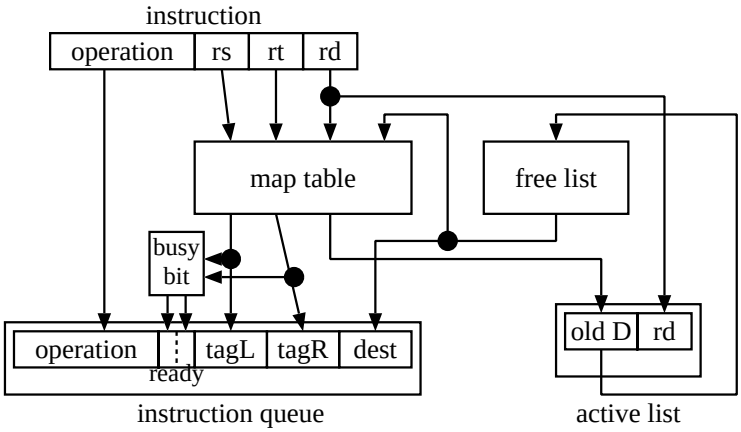

Fig. 1. Instruction window

The instructions remain in the instruction issue queue until their operands have been ready. Once their dependences have been resolved, instruction issue logic schedules the instructions and then issues them into functional units. The

instruction issue queue entries containing the issued instructions are deallocated so that new instructions may be dispatched. We use the term issue to move the instructions from the instruction issue queue to the functional units, where they are executed. After completion of execution, the instructions still wait in the instruction window until their preceding instructions have been retired from the instruction window. When the instructions reach the head of the instruction window, they are retired from it. The instructions may be completed out-of-order but are retired in-order.

The register mapping hardware mainly consists of three structures – a map table, the active list, and a free list. By means of the map table, each logical register is mapped into a physical register. The destination register is mapped to a free physical register which is supplied by the free list, while operand registers are translated into the last mapping assigned to them. The old destination register is kept in the active list. When an instruction is retired, the old destination register which is allocated by the previous instruction with the same logical destination register is freed and is placed in the free list. The translated operand registers are held in the instruction queue as tags which are used for Tomasulo's algorithm. Busy bit table contains a bit indicating whether each physical register contains a valid value. It is used for initializing ready bits in the instruction issue queue for ready operands.

3 Large Instruction Window

While there are a lot of proposals for realizing large instruction windows, we propose an alternative technique. In order to realize large instruction window, the followings are required:

- Deep speculation
- Large instruction issue queue
- Large register file
- Large active list

For each requirement, we propose to utilize the following techniques respectively.

- Selective checkpointing[2, 5]
- Waiting instruction buffer[10]
- Speculative register release
- Folded active list

While the following sections explain them shortly, this paper focuses on the folded active list.

3.1 Selective checkpointing

We can achieve deep speculation by providing a lot of checkpoints. However, every checkpoint requires a plenty of hardware storage. Hence, in order to reduce the number of checkpoints, we utilize to selectively make checkpoints for

predicted branches[2]. Only for branches predicted with low confidence[7], we make checkpoints. This enables deep speculation with relatively small number of checkpoints.

3.2 Waiting instruction buffer

When long latency operations, such as memory operations at cache misses, stall in the pipeline, processor performance is seriously degraded since even ready instructions can not be dispatched into the instruction issue queue. In order for the long latency operations to release the instruction issue queue, we utilize the waiting instruction buffer[10]. Such instructions move to the waiting instruction buffer from the instruction issue queue, and then succeeding ready instructions can be dispatched into the instruction issue queue. This effectively increases instruction issue queue capacity. A drawback in the waiting instruction buffer is that it requires a large active list. To satisfy the requirement, we will propose a large active list in the following section.

3.3 Speculative register release

As explained in the previous section, out-of-order processors utilize register renaming to remove write-after-read and write-after-write hazards. Register renaming requires a large number of physical registers. A physical register is allocated when every instruction is decoded and renamed. Even when the instruction is committed, its associated physical register is not released. It is only released when another instruction who has the same logical register for its destination is committed. This is required for mispredicted branch recovery, while this increases the life time of the register, resulting in the increase in physical register requirement. In order to reduce the number of physical registers, we propose to speculatively release renamed registers with the help of the degree-of-use prediction[4]. The degree-of-use predictor tells us how many consumers will appear for each producer. When the number of consumers matches the predicted value, we speculatively release the producer's renamed register. When mispredicted, processor rolls back to the nearest checkpoint.

3.4 Folded active list

Large monolithic active lists will diminish clock frequency and consume much power. In order to realize a large active list with keeping its speed and power, we propose to construct it with a number of small active lists. We divide a large active list into small pieces of active lists and fold them. The tail of each small active list is logically connected with the head of the next active list.

Figure 2 shows this folded active list. We call the small active list *sublist*. While this structure resembles banking, its operations are different. It also looks like the distributed reorder buffer[9, 13], however, the folded active list is logically a single active list. The key characteristic of the folded active list is that only

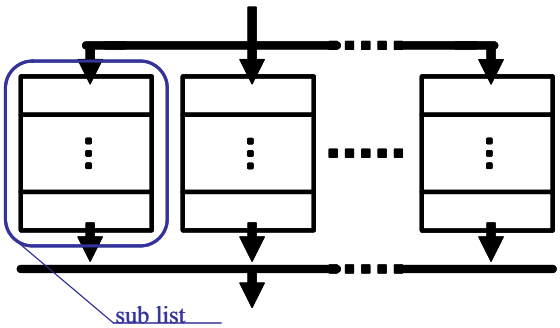

Fig. 2. Folded active list

one sublist is active each for allocation and commitment respectively. That is at most two sublists are active at the time. Please note that there are not any read operations of operand values nor write operations of execution results in the case of the active lists. This is the different characteristic from the reorder buffers. Figure 3 shows how the next new entry is allocated. If the tail of the current sublist is allocated, the current sublist becomes inactive for write. The next sublist becomes active and its head is allocated for the coming instruction. Figure 4 shows an example of the commit operation. In this example, we assume the processor has 4-instruction-wide commit width. Even so, in the folded active list, only two instructions in the current sublist can be committed at the time, and the remaining two instructions are committed at the succeeding cycle after the next sublist is active.

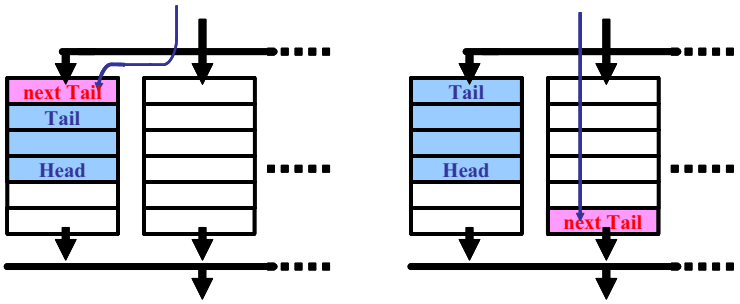

Fig. 3. Restricted allocation

The characteristic that at most two sublists are active at the time keeps high clock frequency and low power. As shown in Figure 5, it is possible to reduce

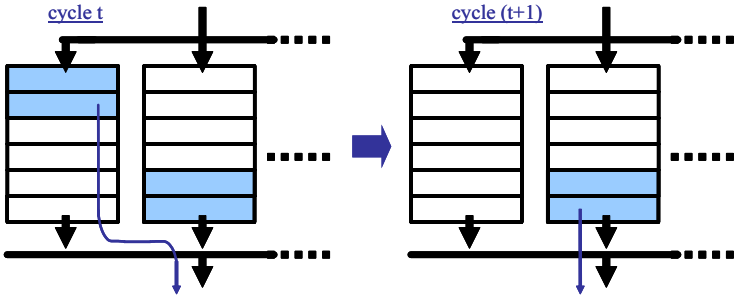

Fig. 4. Restricted commitment

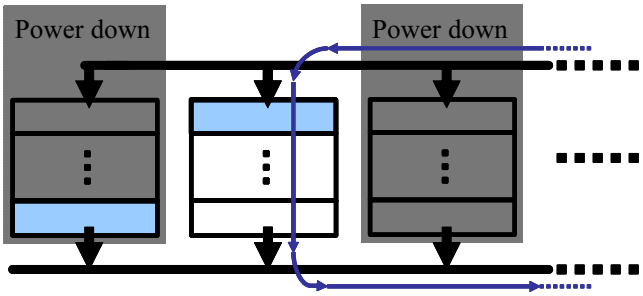

Fig. 5. Selective activation

power supply voltage to inactive sublists. It is also possible to raise threshold voltage to inactive sublist by modulating body bias. This does not diminish processor performance due to the restriction in allocation and commitment. Thus, while the folded active list has a large capacity, its power consumption is comparable to the traditional small active lists.

4 Evaluation Environment

In this section, we describe our evaluation environment by explaining a processor model and benchmark programs.

4.1 Processor model

We implemented a timing simulator using SimpleScalar/PISA tool set (version 3.0)[3]. The configuration of the baseline model is summarized in Table 1.

Table 1. Processor configuration

OoO Core	32-entry instruction issue queue, 32-entry active list, 32-entry load/store queue, 4-wide decode, issue, and retirement
Branch Predictor	16K-entry gshare, 2K 4-way BTB, 64-entry RS, 8-cycle branch misprediction penalty
FUs	4 iALUs, 2 iMUL/iDIVs, 4 fALUs, 2 fMUL/fDIVs
Memory System	32K 4-way L1 I/D caches, 32-byte line size, 3-cycle hit, unified 256K 4-way L2 cache, 32-byte line size, 15-cycle hit, 1,000-cycle memory access, 2 cache ports

4.2 Benchmark programs

The SPEC2000 benchmark suite is used in this study. Table 2 lists the benchmarks and the input sets. We focus on floating-point applications, because we are interested in long latency tolerance. We use the object files distributed at the SimpleScalar LLC web page. For each program, 1 billion instructions are skipped before actual simulation begins. Each program is executed to completion or for 1 billion instructions. We count only committed instructions.

Table 2. Benchmark programs

Program	Input set
171.swim	swim.in
177.mesa	mesa.in mesa.ppm
179.art	c756hel.in a10.img
183.equake	inp.in
188.ammp	ammp.in
301.aspi	–

5 Results

This section presents simulation results. We focus on processor performance. Power efficiency will be improved as explained in Section 3.4. Detailed evaluation on power is remained for the future study.

First, we evaluate how processor performance is improved when we can increase the capacity of the active list with the folded active list. Figure 6 shows the percent increase in processor performance when a 32-entry active list is replaced by a 2K-entry folded active list. The instruction issue queue still has 32

entries. Each sublist is a 32-entry active list, and hence the folded active list evaluated here has 64 sublists. We use instructions per cycle (IPC) as a metric for this evaluation. We can see significant performance improvement, except for `188.ammp`. This is because `188.ammp` does not have the potential of performance improvement. Even when we use the ideal monolithic 2K-entry active list, we do not find any improvement. Even when we remove `179.art` as an exceptional result (over 200% improvement!), an average improvement of 24% is achieved. Since the folded active list maintains clock frequency, this IPC improvement turns out to be net performance improvement.

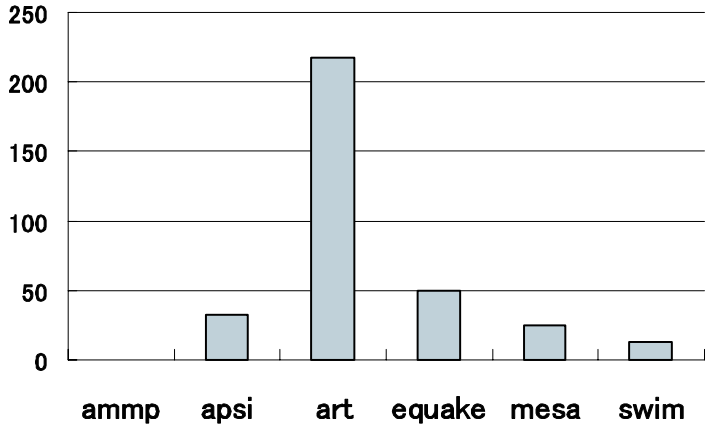

Fig. 6. %Performance improvement over 32-entry monolithic active list

Next, we evaluate how the folding affects processor performance. Figure 7 shows the percent performance loss when we compare the 64x32-entry folded active list with the monolithic but unrealistic 2K-entry active list. We find little performance loss of between 1.3% and 7.5%, except for `188.ammp`, where we find the completely same results for both the folded and monolithic active lists. This performance loss is not serious, because the monolithic active list diminishes clock frequency and thus net performance for both models will be equivalent. In addition, the monolithic active list consumes much higher power than the folded active list, which exploits selective activation.

6 Related Work

Checkpointing for large instruction window is proposed by Cristal et al.[5] and Akkary et al.[2]. This enables to realize a large virtual reorder buffer using a small one and to reduce instruction issue queue entry. Checkpoints are created

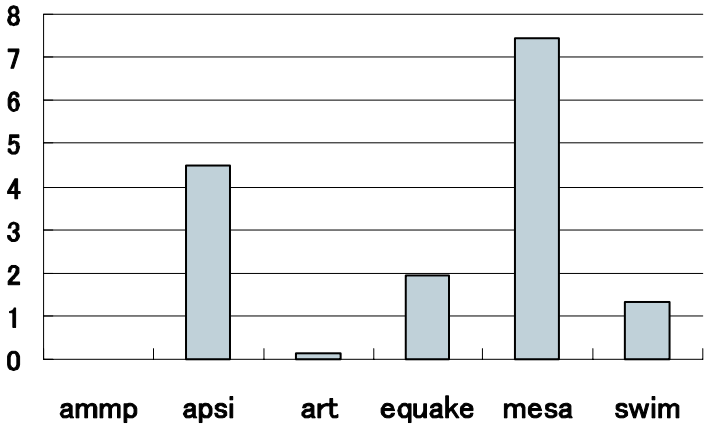

Fig. 7. %Performance loss under 2K-entry monolithic active list

at long latency loads or at low-confidence branches. We follow their studies and determine to take checkpoints at branches because it is shown that branches are a good place for taking checkpoints[5]. Cherry proposed by Martinez et al.[12] also uses checkpointing. However, its purpose is not increasing instruction window but early register release.

Early register release was originally proposed by Moudgill et al.[15]. This uses counters to hold the number of reads. It unfortunately does not support precise exceptions, and this is very severe for modern speculative processors. Monreal et al.[14] and Ergin et al.[6] propose schemes to implement precise exception for early register release. Monreal et al.[14] extends the active list. Ergin et al.[6] uses the Checkpointed Register File bit cell, which is a special register file bit cell and essentially builds a shadow register file.

In order to effectively increase instruction issue queue capacity, a slow lane instruction queue[5], the waiting instruction buffer[10], and a slice processing unit[16] are proposed. The aim is shared by all schemes and is to tolerate long latency memory operations with keeping the instruction issue queue small. Every blocking instruction is moved from the instruction issue queue to a secondary buffer, releasing its instruction issue queue entry for other short latency operations. A good survey on schemes to implement a physically large instruction issue queue is found in [1].

Kucuk et al.[9] and Monferrer et al.[13] propose the distributed reorder buffer for power and temperature reduction. In contrast, the folded active list is proposed to improve processor performance under the constraint of keeping its power consumption.

7 Concluding Remarks

Modern superscalar processors rely on dynamic instruction scheduling for aggregating high performance. Instruction window size is an important factor for exploiting instruction level parallelism. Especially to tolerate long latency operations, large instruction windows are required. In this paper, we proposed a large instruction window, which consists of the selective checkpointing scheme, the waiting instruction buffer, the speculative register release scheme, and the folded active list. The large and fast active list is enabled by folding a conventional large active list, which will diminish clock frequency and consume much power. Based on detailed simulations, we found that the folded active list boosts processor performance by up to more than 200% with the expectation of speed and power, which are equivalent with those of the conventional small active list.

Acknowledgments

This work was supported by PRESTO, Japan Science and Technology Agency, and is partially supported by a Grants-in-Aid for Scientific Research #16300019 from Japan Society for the Promotion of Science.

References

1. J. Abella, R. Canal, and A. Gonzalez, "Power- and Complexity-Aware Issue Queue Designs," IEEE Micro, Volume 23, Number 5, September 2003.
2. H. Akkary, R. Rajwar, and S. T. Srinivasan, "Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors," 36th International Symposium on Microarchitecture, December 2003.
3. D. Burger and T. M. Austin, "The SimpleScalar Tool Set, Version 2.0," ACM SIGARCH Computer Architecture News, Volume 25, Number 3, 1997.
4. J. A. Butts and G. Sohi, "Characterizing and Predicting Value Degree of Use," 35th International Symposium on Microarchitecture, November 2002.
5. A. Cristal, D. Ortega, J. Llosa, and M. Valero, "Kilo-instruction Processors," 5th International Symposium on High Performance Computing, October 2003.
6. O. Ergin, D. Balkan, D. Ponomarev, and K. Ghose, "Increasing Processor Performance Through Early Register Release," 22nd International Conference on Computer Design, October 2004.
7. E. Jacobsen, E. Rotenberg, and J. E. Smith, "Assigning Confidence to Conditional Branch Predictions," 29th International Symposium on Microarchitecture, December 1996.
8. R. Kalla, B. Sinharoy, and J. Tendler, "Simultaneous Multi-threading Implementation in POWER5 – IBM's Next Generation POWER Microprocessor," Hot Chips 15, August 2003.
9. G. Kucuk, O. Ergin, D. Ponomarev, and K. Ghose, "Distributed Reorder Buffer Schemes for Low Power," 21st International Conference on Computer Design, October 2003.
10. A. R. Lebeck, T. Li, E. Rotenberg, J. Koppanalil, and J. Patwardhan, "Large, Fast Instruction Window for Tolerating Cache Misses," 29th International Symposium on Computer Architecture, May 2002.

11. D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton, "Hyper-Threading Technology Architecture and Microarchitecture," Intel Technology Journal, Volume 6, Issue 1, February 2002.
12. J. F. Martinez, J. Renau, M. Huang, M. Prvulovic, and J. Torrellas, "Cherry: Checkpointed Early Resource Recycling in Out-of-order Microprocessors," 35th International Symposium on Microarchitecture, November 2002.
13. P. C. Monferrer, G. Magklis, J. Gonzalez, and A. Gonzalez, "Distributing the Frontend for Temperature Reduction," 11th International Symposium on High-Performance Computer Architecture, February 2005.
14. T. Monreal, V. Vinals, A. Gonzalez, and M. Valero, "Hardware Schemes for Early Register Release," 31st International Conference on Parallel Processing, August 2002.
15. M. Moudgill, K. Pingali, and S. Vassiliadis, "Register Renaming and Dynamic Speculation: an Alternative Approach," 26th International Symposium on Microarchitecture, December 1993.
16. S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, M. Upton, "Continual Flow Pipelines," 11th International Conference on Architectural Support for Programming Languages and Operating Systems, October 2004.
17. K. C. Yeager, "The MIPS R10000 Superscalar Microprocessor," IEEE Micro, Volume 6, Number 2, April 1996.